Introduction to Algorithms

Time Complexity

HELLO!

I am Ahmed Moner

Computer Science Freshman
@E-JUST

You can find me at:
ahmer.moner@ejust.edu.eg

Session Outline:

1. what is Algorithm?
2. Why we need to learn Algorithms?
3. Algorithm Complexity.
4. Good Algorithm?
5. [Worst – Average – Best] Cases.
6. Why the worst case analysis?
7. Asymptotic notation: Big-O.
8. General Rules.
9. Growth Rates.
10. Questions!

“The problem with digital architecture is that an algorithm can produce endless variations, so an architect has many choices.”

“Peter Eisenman”

1. What is Algorithm?

- Set of well-defined instructions for solving a problem or accomplishing a task.
- Skill: The ability to define clear steps to solve a problem

Ex-1: Find the largest number in an array of integers?

Step 01

Initialize a variable (maxNum).

Step 02

Assign its value to the first value of the array.

Step 03

Make a For-loop with a condition.

Step 04

If the number > MaxNum, assign its value to the variable.

Step 05

Now, MaxNum is the latgest number.

2. Why we need to learn Algorithms?

- Algorithmic thinking allows to break down problems solutions in terms of discrete steps.
- You can transform any complicated thing into an algorithm, which will help you in the decision-making process.
- Instead of creating a to-do list, you can write an algorithm to prioritize your daily tasks.

3. Algorithm Complexity

- Knowing the complexity of the algorithms allows you to answer questions like:
  - ✓ How long the program run on an input?
  - ✓ How much space will it take?
  - ✓ Is the problem will be solved?

Time Complexity

The amount of time by an algorithms to execute.

Space Complexity

The amount of space or memory needed to run the program.

4. Good Algorithms:

I. Efficient.
   I. Running time, solve problem in the least possible steps.
   II. Space used, take space as less as possible.

II. Efficiency.
   Relative with the number of input’s bits.

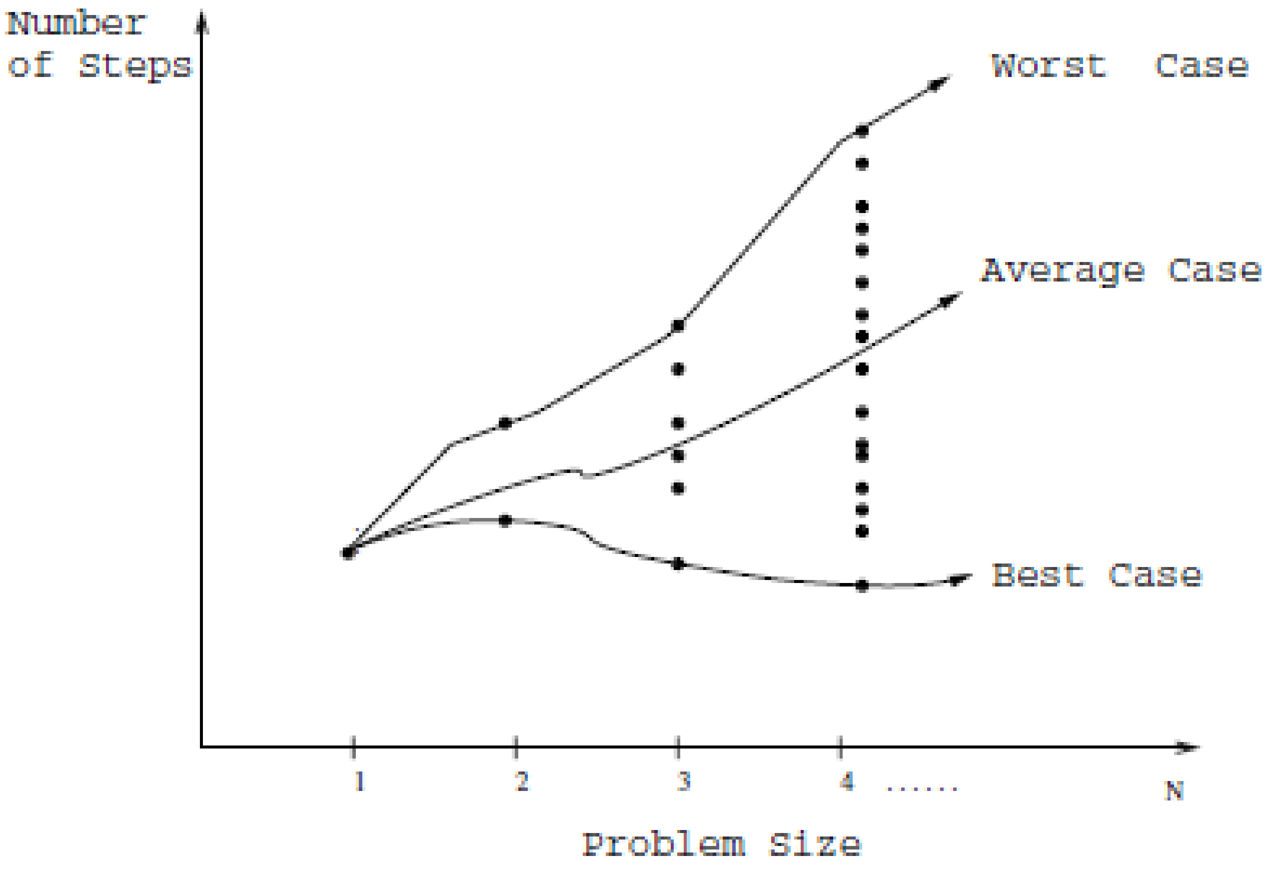
5. [Worst – Average – Best] Cases.

If you need to sort a list of numbers descending:

1. Best-case: They are already sorted,
   and the pointer only checks them.

2. Worst-case: They are already sorted as well,
   but in ascending sort,
   averaged over all possible inputs.

3. Average-case: some numbers sorted and some are not.



Number
of Steps
Worst Case
Average Case
Best Case
1
2
3
4 ......
N
Problem Size

6. Why the worst case analysis?

- It’s the longest running time of any input of size n, the upper bound.
- We usually concentrate on finding only the worst case.
- We always deal with this scenario, as the algorithm operations will never make more than it.

7. Asymptotic notation: Big-O.

- A special notation tells you how fast an algorithm is, the number of operations should be done.
- Algorithm speed isn’t measured in seconds, but the growth of the number of operations.
- Running time increases as the size of a list increases.

8. General Rules

Ignore all the following:

- The lower bound terms
- The base of the logarithm
- The coefficients, especially of the highest-order terms

9. Growth Rates

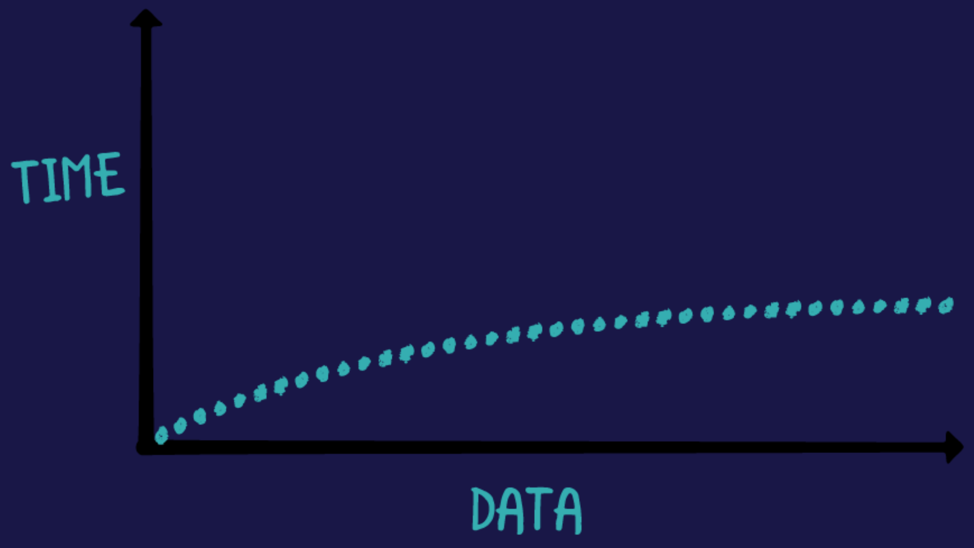
- The rate at which the cost of the algorithm grows as the size of its input grows.

- A. Constant Time (1): O (1)
  - A. 1 doesn't mean second nor instant.
  - B. The time taken will still the same, regardless of how big the list is.

B. Logarithmic Time (2): O (log (n))

A. Any code that use binary search.

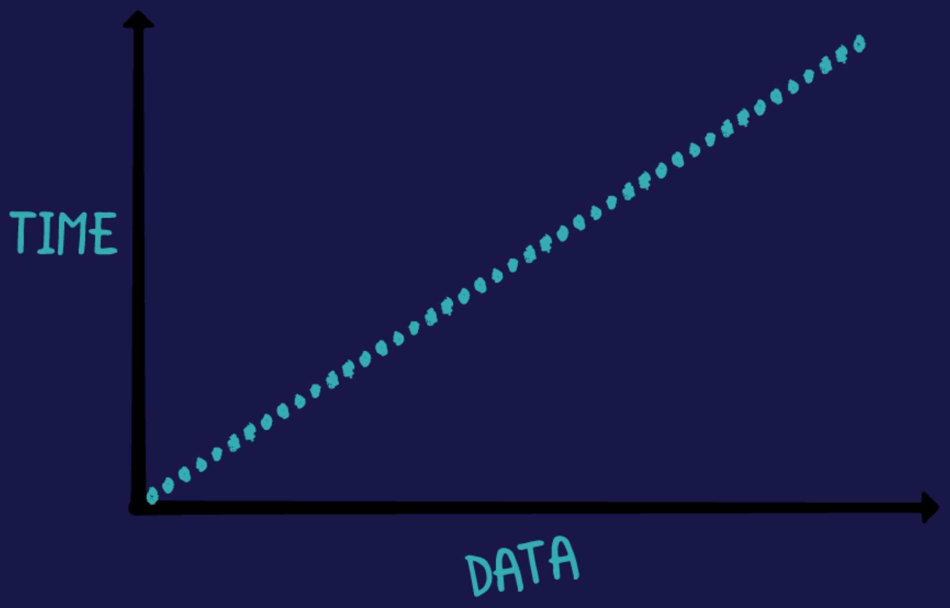
B. You are eliminating the half of the remaining steps every time.
So, it will take log(n) of base 2
to reach the desired number.



C. Linear Time (3): O (n)

A. Any code that use simple search.

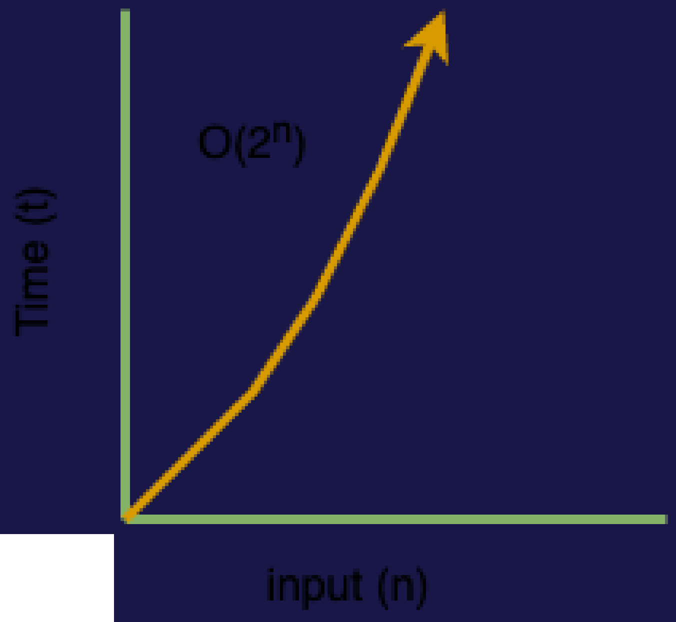
B. You aren't eliminate any step every time.
So, it will take log(n) of base 2
to reach the desired number.



D. Exponential Time (4): O (n^2)

A. Any code that has nested for loops.

B. You are eliminating the half of the remaining steps every time.
So, it will take log(n) of base 2
to reach the desired number.



- For x in range (0, n); //O(N)
  - Print x;

- For x in range(0, n); //$O(N^2)$
  - For y in range (0, n);
    - Print x * y; // O(1)

- Three nested for loops? $O(N^3)$ Four? $O(N^4)$ and so on

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

- X = 10 + (5 * 2);
- Y = 20 - 2;
- System.out.print(”x + y”);

- Total time is = O(1) + O(1) + O(1) = 3 * O(1)

O (1)

```
for(i= 0 ; i < n; i++){
   for(j = 0; j<n ;j++){
      cout<< i << " ";
   }
}
```

O (n)

```
for(i= 0; i < n; i++){
   for(j = 1; j < n; j = j*2){
      cout << i << " ";
   }
}
```

$$O(n * log_2(n))$$

Javascript

```
for(var i=0;i<n;i++)
    i*=k
```

1. O(n)
2. O(k)
3. $O(log_k n)$
4. $O(log_n k)$

$$O(log_k(n))$$

CPP

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

Options:

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

$$O(log_2(n))$$


Solve With
<Mentors>

1.

```cpp
void printAllItemsTwice(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << '\n';
    }
    for (int i = 0; i < size; i++) {
        cout << arr[i] << '\n';
    }
}
```

O (n)

2.

```cpp
void printFirstItem_FirstHalf_SayHi100Times(int arr[], int size) {
    cout << "First element of array" << arr[0] << '\n';
    for (int i = 0; i < size / 2; i++) {
        cout << arr[i] << 'n';
    }
    for (int i = 0; i < 100; i++) {
        cout << "Hi\n";

    }
}
```

O (n)

3.

```
void printAllNumbersThenAllPairSums(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << '\n';
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << arr[i] + arr[j] << '\n';
        }
    }
}
```

$O(n^2)$

4.

```
int main() {
    int a = 0, b = 0;
    int N = 4, M = 4;
    for (int i = 0; i < N; i++) {
        a = a + 10;
    }
    for (int i = 0; i < M; i++) {
        b = b + 40;
    }
    cout << a << ' ' << b;
    return 0;
}
```

O (n+m)

5.

```
int main() {
    int a = 0, b = 0;
    int N = 4, M = 5;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            a = a + j;
            cout << a << ' ';
        }
        cout << endl;
    }
    return 0;
}
```

O (n*m)

6.

```
int main() {
    int N = 8, k = 0;
    for (int i = N / 2; i <= N; i++) {
        for (int j = 2; j <= N; j = j * 2) {
            cout << k << ' ';
            k = k + N / 2;
        }
    }
    return 0;
}
```

$O(n * log_2(n))$

7.

```cpp
int main() {
    int N = 18;
    int i = N, a = 0;
    while (i > 0) {

        cout << a << ' ';
        a = a + i;
        i = i / 2;
    }
    return 0;
}
```

$O(log_2(n))$

9.

```cpp
int fun(int n) {
    int i, j, k, p, q = 0;
    for (i = 1; i < n; ++i) {
        p = 0;
        for (j = n; j > 1; j /= 2)
            ++p;
        for (k = 1; k < p; k *= 2)
            ++q;
    }
    return q;
}
```

$O(n * log_2(n))$

8.

```cpp
int y = 0;
for (int j = 1; j * j <= n; j++)
    y++;
```

$O(\sqrt{n})$

"A person who never made a mistake never tried anything new."

Bonus Question

4. Work out the computational complexity (in the "Big-Oh" sense) of the following piece of code and explain how you derived it using the basic features of the "Big-Oh" notation:

```
for( int bound = 1; bound <= n; bound *= 2 ) {
    for( int i = 0; i < bound; i++ ) {
        for( int j = 0; j < n; j += 2 ) {
            ... // constant number of operations
        }
        for( int j = 1; j < n; j *= 2 ) {
            ... // constant number of operations
        }
    }
}
```

Answer [Bonus Question]

4. The first and second successive innermost loops have $O(n)$ and $O(\log n)$ complexity, respectively. Thus, the overall complexity of the innermost part is $O(n)$. The outermost and middle loops have complexity $O(\log n)$ and $O(n)$, so a straightforward (and valid) solution is that the overall complexity is $O(n^2 \log n)$.

THANKS!

ANY QUESTIONS?